

Übung zu Betriebssystembau (Ü BS)

Interruptbehandlung in OO-Stubs

Daniel Lohmann

Lehrstuhl für Informatik IV

WS 06-07



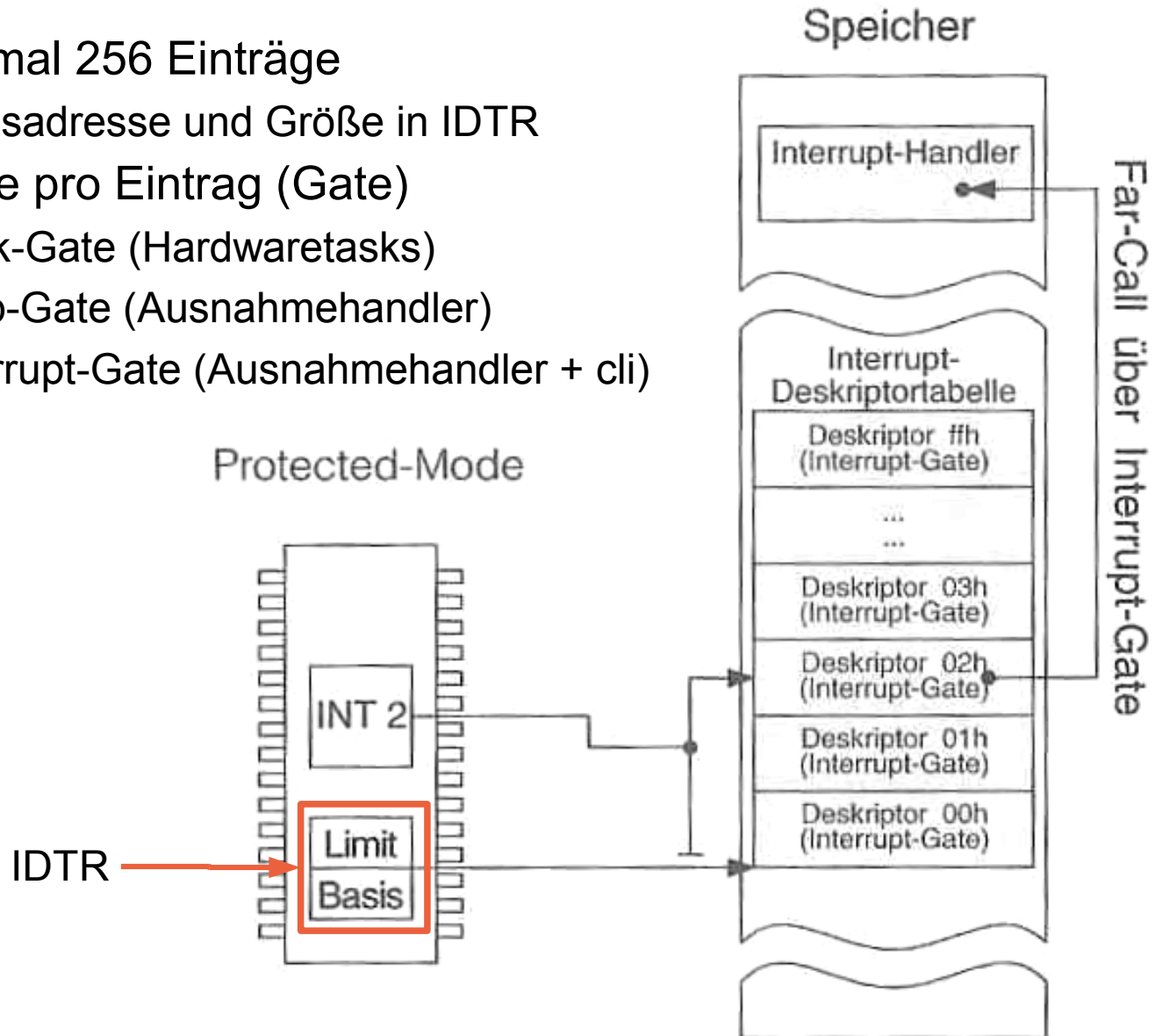
Agenda: IRQ-Behandlung in OO-Stubs

- Interrupts und Traps bei x86
 - Die Interrupt-Deskriptor-Tabelle (IDT)
 - Aufbau der IDT
 - Traps und Hardware IRQs
- Der Programmierbare Interruptcontroller PIC 8295A
 - Aufbau
 - Verwendung im PC
 - Initialisierung und Programmierung
- Interruptbehandlung in OO-Stubs
 - Ablauf
 - Kontextsicherung

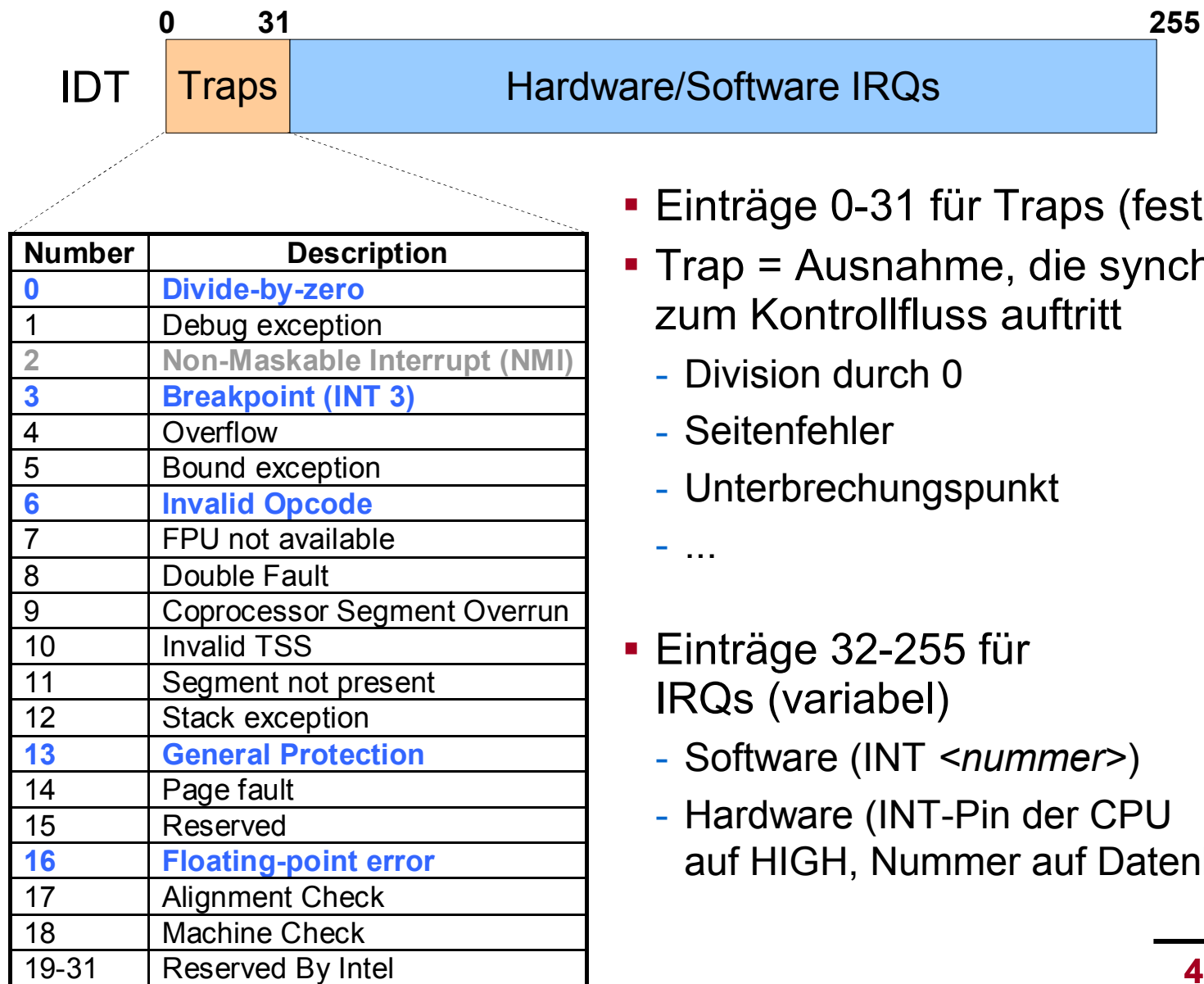


i386 Interrupt-Deskriptortabelle (IDT)

- maximal 256 Einträge
 - Basisadresse und Größe in IDTR
- 8 Byte pro Eintrag (Gate)
 - Task-Gate (Hardwaretasks)
 - Trap-Gate (Ausnahmehandler)
 - Interrupt-Gate (Ausnahmehandler + cli)



i386 IDT: Aufbau



- Einträge 0-31 für Traps (fest)
- Trap = Ausnahme, die synchron zum Kontrollfluss auftritt
 - Division durch 0
 - Seitenfehler
 - Unterbrechungspunkt
 - ...
- Einträge 32-255 für IRQs (variabel)
 - Software (INT <nummer>)
 - Hardware (INT-Pin der CPU auf HIGH, Nummer auf Datenbus)



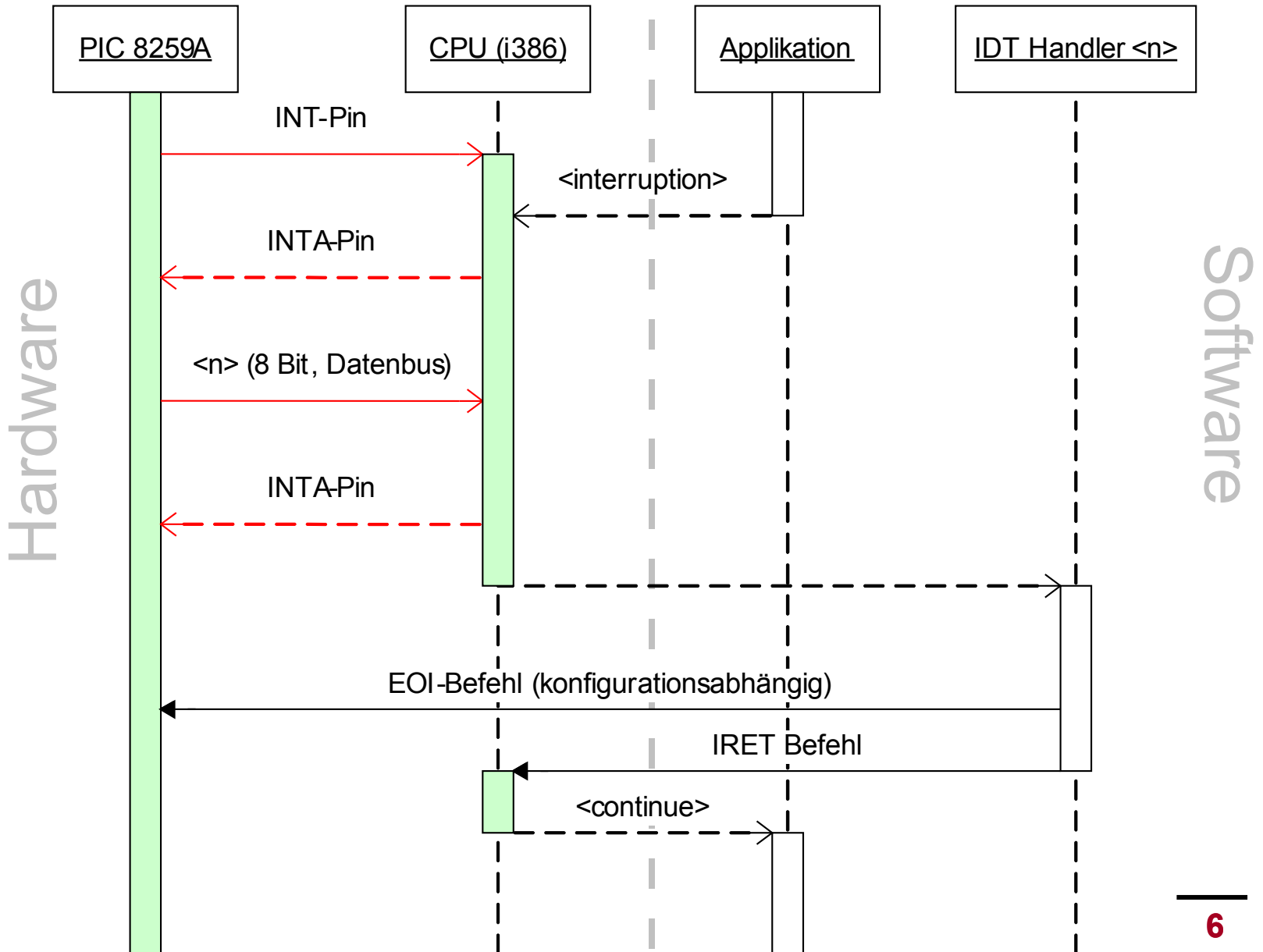
Hardware-IRQs bei x86 CPUs

- Bis einschließlich i486 hatten x86 CPUs nur **einen** Interrupt-Eingang (INT) plus einen NMI Eingang
 - INT kann mit dem IE-Bit im Statuswort maskiert werden
 - cli-Befehl (clear interrupt enable) – Interruptverarbeitung sperren
 - sti-Befehl (set interrupt enable) – Interruptverarbeitung freigeben
 - NMI kann auf der CPU nicht maskiert werden (sagt ja schon der Name)
 - beim PC aber trotzdem durch externe Hardware...

- Externer Controller muss Nummer auf den Bus legen
 - Beim PC ist das der **Programmable Interrupt Controller 8259A**
 - Datenaustausch zwischen CPU und PIC 8259A erfolgt nach einem festgelegtem Protokoll



Ablauf eines Hardware IRQ



Zustandssicherung

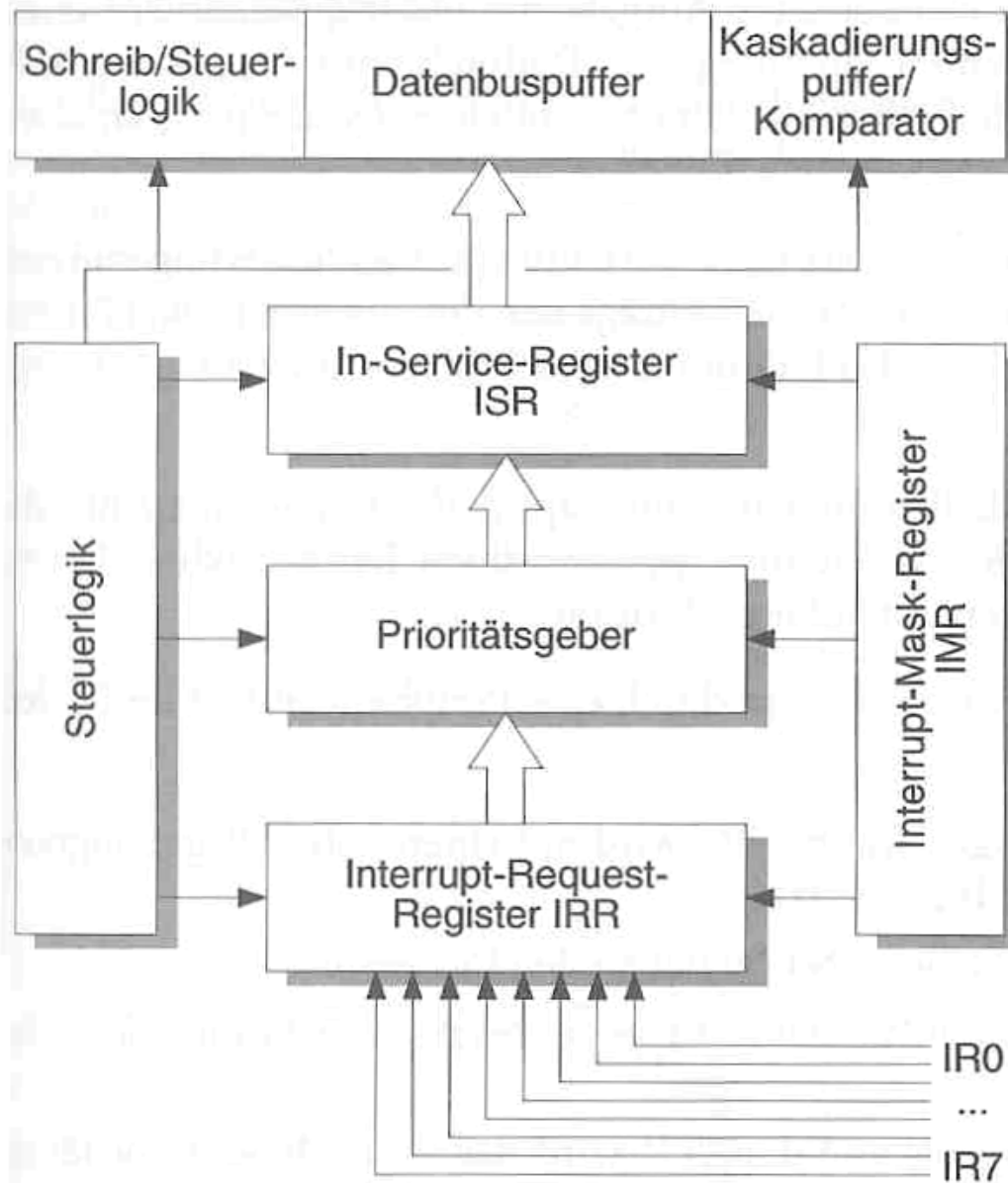
- Wenn eine Unterbrechung eintritt, sichert die CPU automatisch einen Teil des Zustands auf dem Stapel
 - condition codes (eflags)
 - aktives Codesegment (cs)
 - Rücksprungadresse (eip)



- Der automatisch gesicherte Zustand wird bei der Ausführung von iret zurückgesetzt
 - verwendet der Handler weitere Register, so muss er diese selber sichern!



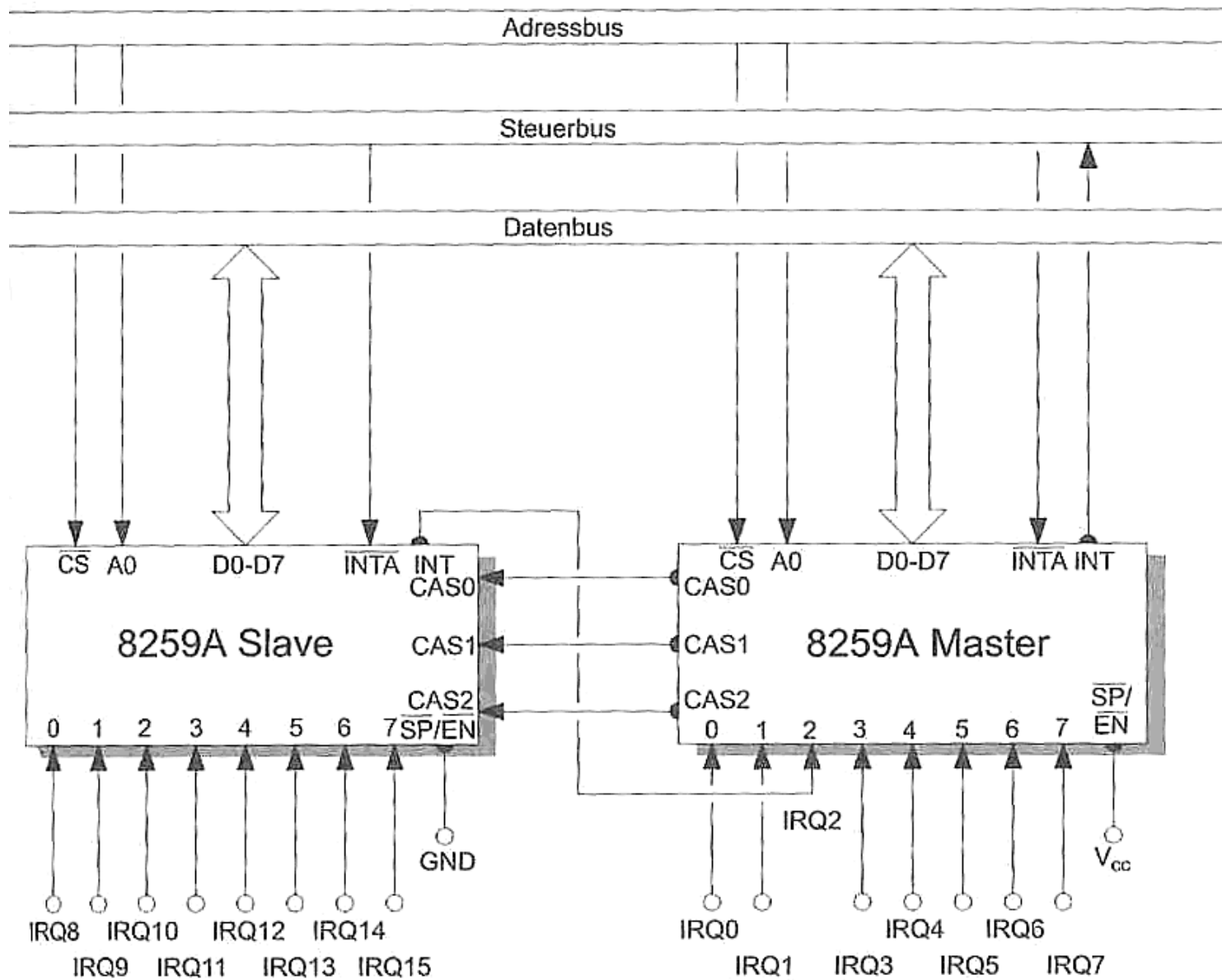
PIC 8259A – Interner Aufbau



höchste

niedrigste

Kaskadierung im PC (15 Interrupts)



Zugriff auf die PICs über IO-Ports

- Jeder PIC hat zwei Ports, die gelesen/geschrieben werden können
- Schreibdaten: ICW1-4, OCW1-3
 - ICW = Initialization Control Word – Initialisierung des PICs
 - OCW = Operation Control Word – Kommandos im Betrieb
- Lesedaten abhängig von Kommando

Lesen

IRR
ISR
Offset

IMR

Master

Port 0x20

Port 0x21

Slave

Port 0xa0

Port 0xa1

<wie Master>

Schreiben

ICW1 (Initialisierung beginnen)
OCW2 (EOI, ...)
OCW3 (IRR lesen, ISR lesen, ...)

ICW2-4 (Initialisierungsdaten)
OCW1 (= IMR)

<wie Master>



Initialisierung der PICs – Teil 1

ICW1

7	6	5	4	3	2	1	0
0	0	0	1	LTIM	0	SNGL	IC4

LTIM: 0=Flankentriggerung

SNGL: 0=kaskadierte PICs

IC4: 0=kein ICW4

1=Pegeltriggerung

1=nur Master

1=ICW4 notwendig

OO-Stubs Einstellung:

Master

Slave

00010001

00010001

ICW2

7	6	5	4	3	2	1	0
Off7	Off6	Off5	Off4	Off3	0	0	0

Off7..Off3: programmierbarer Offset des Interrupt-Vektors

OO-Stubs Einstellung:

Master

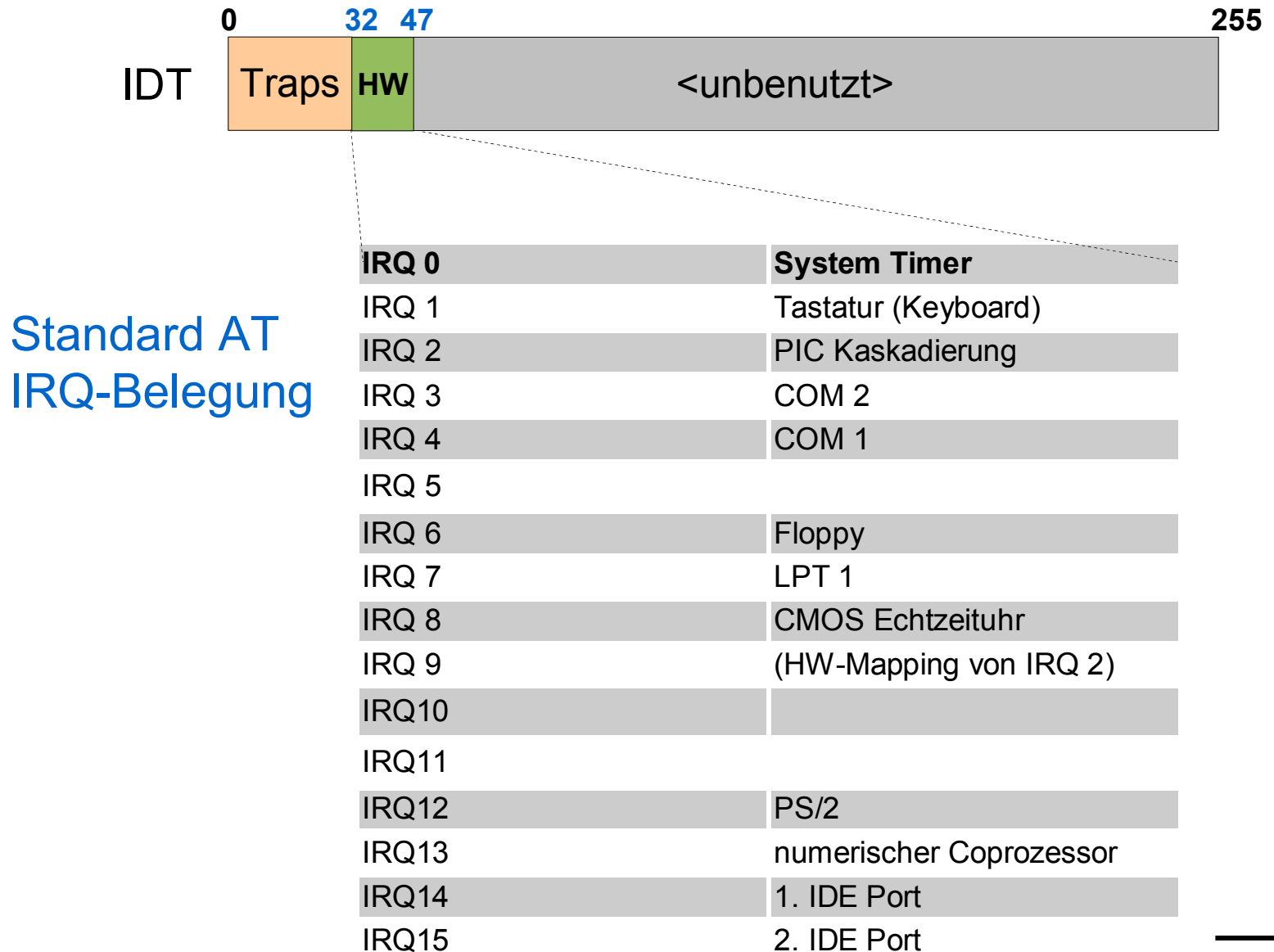
Slave

00100000

00101000



Mapping der HW-IRQs (OO-Stubs)



Initialisierung der PICs – Teil 2

ICW3 (Slave)

7	6	5	4	3	2	1	0
0	0	0	0	0	ID2	ID1	ID0

ID2..ID0: Identifizierungsnummer des Slave-PIC

ICW3 (Master)

7	6	5	4	3	2	1	0
S7	S6	S5	S4	S3	S2	S1	S0

S7..S0: 0=zugehörige IR-Leitung ist mit Peripheriegerät verbunden oder frei
1=zugehörige IR-Leitung ist mit Slave-PIC verbunden

ICW4

7	6	5	4	3	2	1	0
0	0	0	SF NM	BUF	M/S	AEOL	μPM

SFNM: 0=kein Special-Fully-Nested-Modus

BUF: 0=kein gepufferter Modus

M/S: 0=Slave-PIC

AEOL: 0=manueller EOI

μPM: 0=Betrieb im MCS-80/85-Modus

1=Special-Fully-Nested-Modus

1=gepufferter Modus

1=Master-PIC

1=automatischer EOI

1=Betrieb im 8086/88-Modus

OO-Stubs Einstellung:

Master

00000100

Slave

00000010

OO-Stubs Einstellung:

Master

00000011

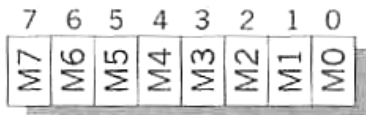
Slave

00000011



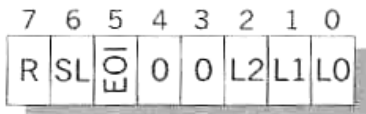
Programmierung der PICs

OCW1



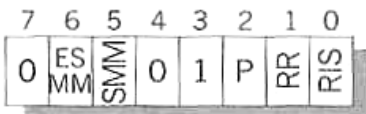
M7..M0: 0=zugehörige IRQ-Leitung ist nicht maskiert
1=zugehörige IRQ-Leitung ist maskiert

OCW2



000: im AEOI-Modus rotieren
001: nicht-spezifischer EOI-Befehl
010: kein Vorgang (NOP)
011: spezifischer EOI-Befehl (mit L2..L0)
100: im AEOI-Set-Modus rotieren
101: bei nicht-spezifischem EOI-Befehl rotieren
110: Prioritätsbefehl setzen
111: bei spezifischem EOI-Befehl rotieren

OCW3



ESMM, SMM:	00=kein Vorgang (NOP)	01=kein Vorgang (NOP)
	10=spez. Maske löschen	11=spez. Maske setzen
RR, RIS:	00=kein Vorgang (NOP)	01=kein Vorgang (NOP)
	10=IRR lesen	11=ISR lesen
P:	Polling: 0=kein Polling	1=Polling-Modus

- Interruptmaske (IMR)
 - schreiben und lesen über Port 0x21 / 0xa1



Interrupthandler in OO-Stubs

- Behandlung startet in der Funktion `guardian()`

- bekommt die IRQ-Nummer als **Parameter**:

```
void guardian( unsigned int slot ) {  
    ... // IRQ-Handler (Gate) aktivieren  
}
```

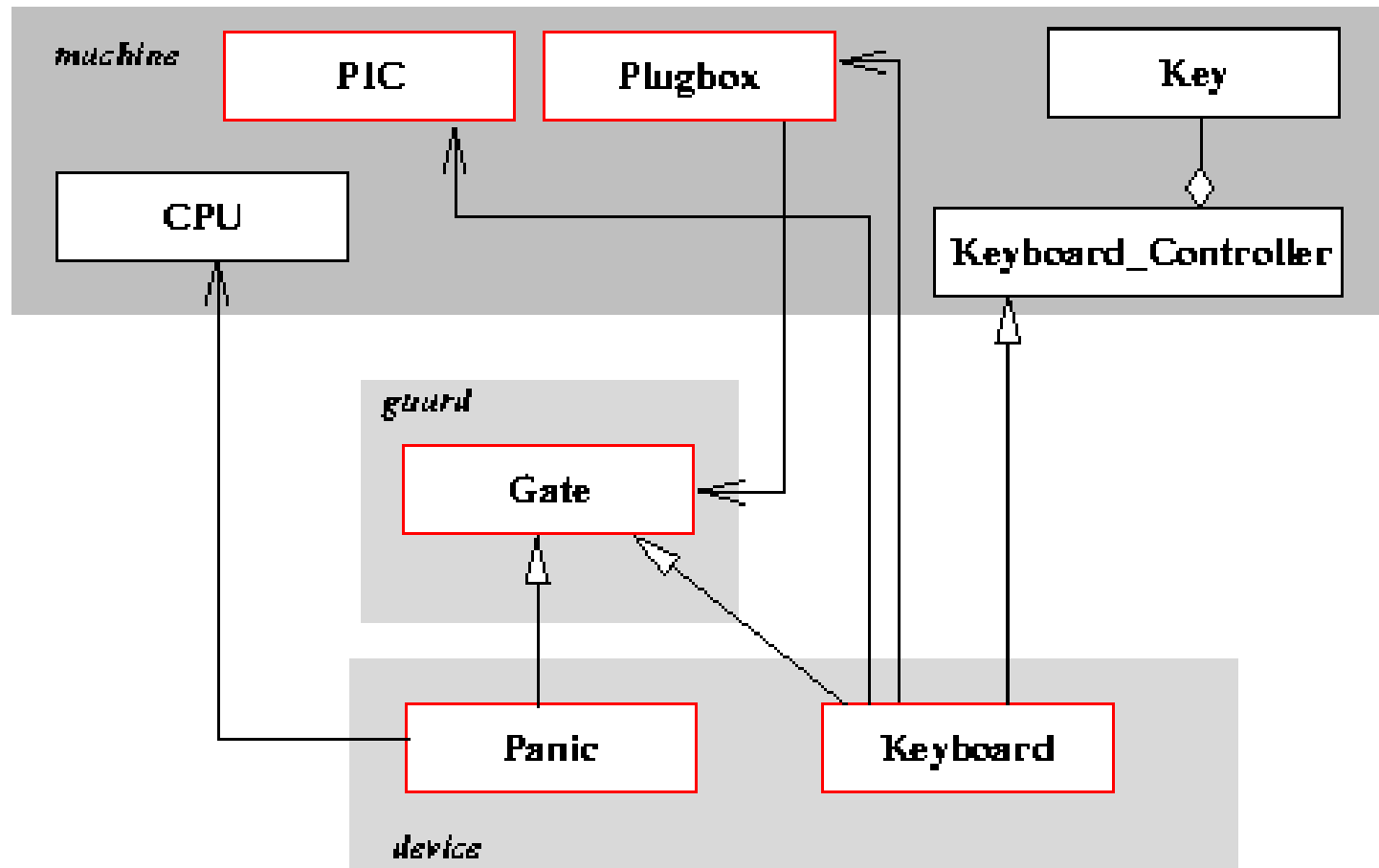
- Interrupts sind während der Abarbeitung gesperrt
 - können mit `sti` manuell wieder zugelassen werden
 - werden automatisch wieder zugelassen, wenn `guardian()` terminiert

- Die eigentlichen (spezifischen) IRQ-Handler

- sind Instanzen der Klasse *Gate*
- werden an- und abgemeldet über die Klasse *Plugbox*



Interrupthandler in OO-Stubs



C-Funktionen als IRQ-Handler

- `guardian()` ist eine C-Funktion
 - erwartet die IRQ-Nummer als Parameter
- Kann `guardian()` direkt in die IDT eingetragen werden?
 - Wer sichert den Kontext?
 - Wie wird der Parameter übergeben?
 - Wo steht das `iret` zum Abschließen der IRQ-Behandlung?

⇒ Assembler-Stubs in *startup.asm*



Kontextsicherung

- Kontextsicherung:
 - eflags, cs und eip wurden bereits von der CPU gesichert
 - alle weiteren Register müssen vom IRQ-Handler gesichert werden
 - entweder im Assembler-Teil
 - oder der Compiler generiert bereits entsprechenden Code

- Kontextsicherung beim Aufruf von Funktionen
 - Lösung 1: *Aufrufende* Funktion sichert alle Register, die sie später noch braucht
 - Lösung 2: *Aufgerufene* Funktion sichert alle Register, die sie verändert
 - Lösung 3: Ein Teil der Register wird vom Aufrufer, ein anderer Teil vom Aufgerufenen gesichert



Kontextsicherung in Hochsprachen

- In der Praxis wird Lösung 3 verwendet
 - Grundsätzlich hängt das natürlich vom Compiler ab
 - CPU-Hersteller definiert jedoch Konventionen, damit Interoperabilität auf Binärcodeebene sichergestellt ist
- Register werden in 2 Subsets aufgeteilt
 - Flüchtige Register („scratch registers“)
 - Compiler geht davon aus, dass Unterprogramm den Inhalt verändert
 - Aufrufer muss Inhalt gegebenenfalls sichern
 - bei x86 sind **eax**, **ecx**, **edx** und **eflags** als flüchtig definiert
 - Nichtflüchtige Register („non-scratch registers“)
 - Compiler geht davon aus, dass der Inhalt nicht verändert wird
 - Aufgerufene Funktion muss Inhalt gegebenenfalls sichern
 - bei x86 sind alle sonstigen Register als nicht-flüchtig definiert

Interrupt-Handler müssen auch flüchtige Register sichern!

